Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Lab 0 is due this Friday
  - Not graded and no deliverables
- Recitations start next week
- Homework 1 will be posted this Friday on GradeScope
- Available on Canvas
  - JDB Example
  - Video on debugging using VS Code
  - Link to Draft slides and handouts repository

Recall from previous lecture

A technique for modeling runtime of algorithms

- $\sum_{all\ statements} Cost\ of\ statement * frequency\ of\ statement$

- Split the algorithm into blocks such that
  - the code statements in each block have the same frequency
- $\sum_{all\ blocks} Costof\ block * frequency\ of\ block$

Example 1

```
public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i <    i++) {
        sum
    }
    retu
}
```

- How much time does that statement take?
- Depends on machine used, other programs running, etc.
- Let’s assume it is a constant C_0

Example 1

```
public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
    }
}
```

- Cost = C_0
- How many times does that statement execute in one run of the algorithm?
- Just once!

Example 1

```
public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i < n; i++) {

    }
}
```

- Cost = C_0
- frequency = 1

Example 1

```
public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + a[i];
    }
    retu
}
```

- Cost = C_1
- frequency = 1

Example 1

```
public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + a[i];
    }
    return sum;
}
```

- Cost = C_2
- frequency = 1

Example 1

```
public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + a[i];
    }
    return sum;
}
```

- Cost = C_0
- frequency = 1

Example 1

```
public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + a[i];
    }
    return sum;
}
```

- Cost = C_1
- frequency = n

Example 1

```
public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + a[i];
    }
    return sum;
}
```

- Cost = C_2
- frequency = n+1

Algorithm Analysis Example 1

$$\text{for}(\overbrace{\boxed{i=0}}^{1};\ \overbrace{\boxed{i<n}}^{n+1};\ \overbrace{i{+}{+}}^{n})$$

$$a[i] = i;$$

What is the running time?

- $\sum_{all\ blocks} Costof\ block * frequency\ of\ block$
- $= C_0 * 1 + C_1 * n + C_2 * (n+1)$

```
public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + a[i];
    }
    return sum;
}
```

Algorithm Analysis Example 2

```
public int sum(int[] a, int x) {
  int sum = 0;
  if(x > 0){
     for (int i = 0; i < n; i++) {
        sum = sum + a[i];
     }
  }
  return sum;
}
```

Algorithm Analysis Example 2

1

```
if( x > 0 ){
  for( i=0; i<n; i++)
    a[i] = i;
```

0 or 1

0 or n

2

Algorithm Analysis Example 3

$$\overset{1}{}$$

$$\text{for}(\boxed{i = n};\ i >= 1;\ i = i/2)$$

$$a[i] = i;$$

$$\log n$$

What is the runtime of ThreeSum?

```
public static int count(int[] a) {
    int n = a.length;
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            for (int k = j+1; k < n; k++) {
                if (a[i] + a[j] + a[k] == 0) {
                    cnt++;
                }
            }
        }
    }
```

frequency: $f_0 = 1$
cost: t_0

freq: $f_1 = n$
cost: t_1

$f_4 = x$ (the number of triples that sum to 0 in the input array)

$0 \leq x \leq C(n, 3)$

cost: t_4

What is the runtime of ThreeSum?

frequency: $f_0 = 1$
cost: t_0

freq: $f_1 = n$
cost: t_1

$f_2 = (n-1) + (n-2) + (n-3) + \ldots + 1$
$= \frac{n-1}{2}(n - 1 + 1) = \frac{n^2}{2} - \frac{n}{2}$
cost = t_2

$f_4 = x$ (the number of triples that sum to 0 in the input array)
$0 \leq x \leq C(n,3)$
cost: t_4

$f_3 = C(n,3) = n_{C_3} = \frac{n!}{(n-3)!3!}$
$= \frac{n(n-1)(n-2)(n-3)!}{(n-3)!6} = \frac{n(n-1)(n-2)}{6} = \frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3}$
cost: t_3

Grand total = $\sum_{i=0}^{4} f_i * t_i$

$$= \frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0 + t_4 x$$

What is the runtime of ThreeSum?

$$\frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0 + t_4x$$

- Remember that $0 \le x \le C(n, 3)$
- If x = 0 → best-case runtime

$$\frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0$$

- If x = $C(n, 3)$ → worst-case runtime

$$\frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0 + t_4(\frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3})$$

Algorithm Analysis

- You see that this analysis can get ugly at times
- Do we really need to consider all these terms and constants?
  - The answer is No!

Enter Asymptotic Analysis

Algorithm Analysis

- Determine *resource usage* as a function of *input size*
  - e.g., *n,* in 3-sum, the length of the array size, is the input size
  - We already did that for ThreeSum
- Measure ***asymptotic*** performance
  - Performance as input size increases to infinity

Asymptotic performance

Focus on the **order of growth** of functions**,** not on exact values

Asymptotic performance

Order of growth captures how fast the function value increases when the input increases; in particular, for a function *T(n)*

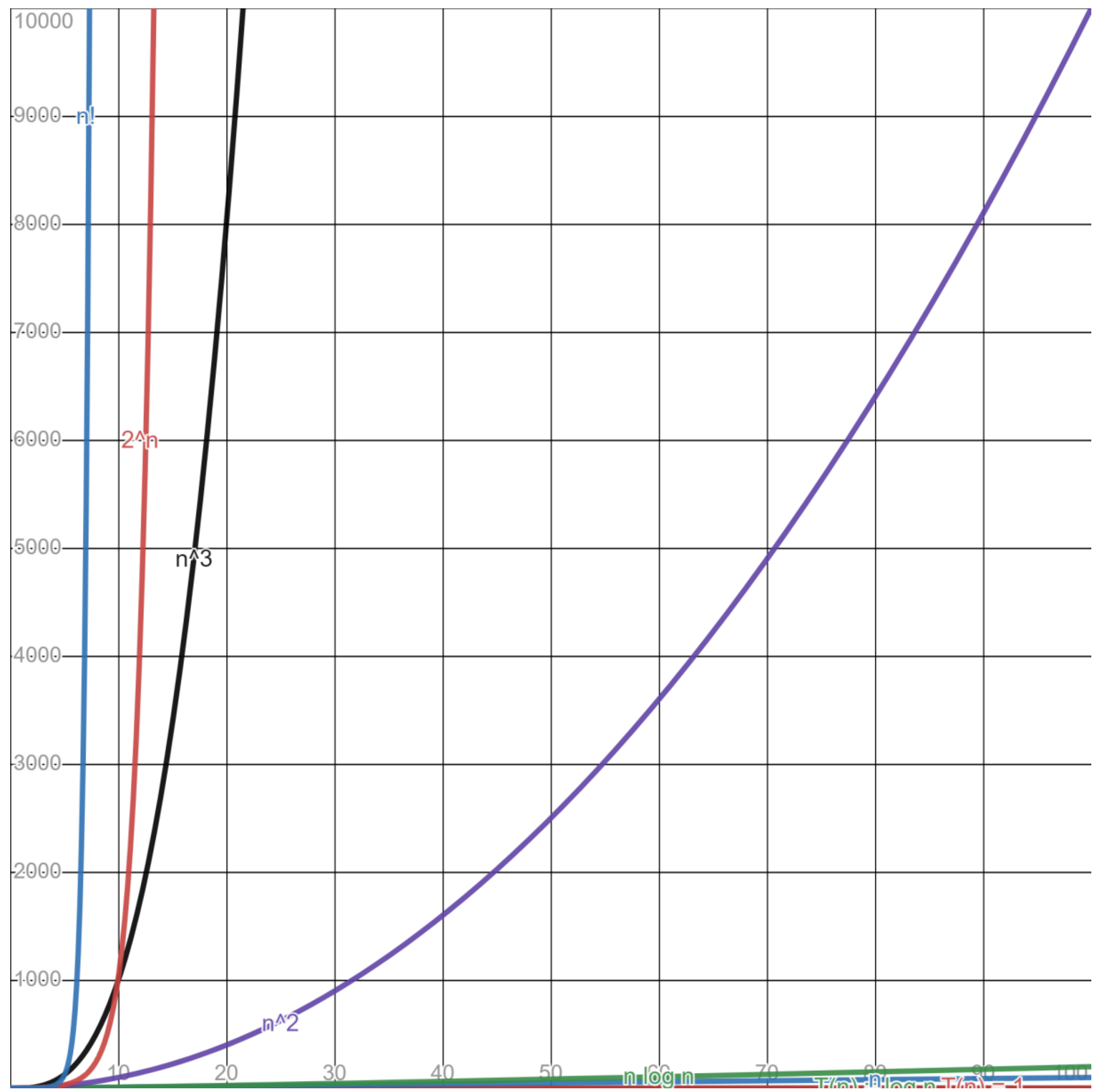
- When *n* doubles, does *T(n)* essentially
  - stay constant
  - increase by a constant
  - double as well
  - quadruple (**x**4)
  - increase eightfold (**x**8)
  - … ?
- When *n* increases by 1, does *T(n)* essentially
  - double
  - increase n-fold (**x**n)
  - … ?

Asymptotic performance

We don’t care as much about the exact value of *T(n)*

Common orders of growth in Algorithm Analysis

- Constant - 1
- Logarithmic - log n
- Linear - n
- Linearithmic - n log n
- Quadratic - n^2
- Cubic - n^3
- Exponential - 2^n
- Factorial - n!



https://www.desmos.com/calculator/tgud0bb1mz

Side note

What does $log_2 n$ really mean?

$log_2 n$ is the number of times n can be divided by 2 before until we reach 1 or less

Side note

Why do we use $T(n)$ instead of $f(x)$?

T stands for Time, or running time

Using n signifies that the input is a positive integer

Order of growth of runtime functions

- For runtime functions, is it better to have a function with a high order of growth or a low order of growth?
  - low order of growth means when input size increases, the value of the runtime won't increase by much
  - This means a fast algorithm
  - So, we want a low order of growth function for runtime

Quick algorithm analysis

- How can we determine the order of growth of a function?
  - Ignore lower-order terms
  - Ignore multiplicative constants
- Example: polynomial functions
  - $\mathrm{T(n)} = 5n^3 + 53n + 7$
  - Terms: $5n^3, 53n, 7$
  - $5n^3$ is of order 3, $53\ n$ is of order 1
  - what is the order of the term 7?

Example

$$5n^3 + 53n + 7 \ \rightarrow n^3$$

- Warning: this is a simplification
  - It works for most of the algorithms in this course
- In some cases, it is difficult to determine the highest-order term
- In some cases, the constant factors play a significant role
  - e.g., small or medum-size input and large constant factors

But …

- Can we say $5n^3 + 53n + 7 = n^3$?
- No! We need a mathematical notation
- $5n^3 + 53n + 7 = O(n^3)$
  - Read as Big O of n^3
- It means the order of growth of $5n^3 + 53n + 7$ is no more than ($\leq$) the order of growth of n^3

Notations

- May also see:
  - $f(x) \in O(g(x))$ or
  - $f(x) = O(g(x))$
- used to mean that $f(x)$ is $O(g(x))$
- Same for the other functions

Notations

$\Theta(n)$

$10n$

$\log n$

$50n+1$

$3\sqrt{n}+10$

7

n^2

$10n \in \Theta(n)$

$\log n \in \Theta(n)$

$n^2 \notin \Theta(n)$

The Big O Family

- O roughly means ≤
  - Big O
- o roughly means <
  - Little O or O-micron
- Ω roughly means ≥
  - Big Omega
- ω roughly means >
  - Little Omega
- Θ roughly means =
  - Theta
- Relationships are between orders of growth, not between exact values!

Asymptotic analysis approximations

- How can we determine the order of growth of a function?
  - Ignore lower-order terms
  - Ignore multiplicative constants
- Would it matter if the frequency of a statement is *n* or *n+1?*
  - *No!*
- Would it matter if it is *n* or *2n?*
  - *No!*
- Would it matter if it is 2^n or 2^{2n}?
  - Yes! Why?

A couple useful approximations under Asymptotic Analysis

$\sum$ arithmetic Series:

$$1 + 2 + 3 + 4 + \dots + n$$

$$= \#terms \left(\frac{\text{first term} + \text{last term}}{2} \right)$$

$$= \Theta(\#terms * \text{largest term})$$

$\sum$ geometric Series:

$$1, 2, 4, 8, 16, \dots$$

$$1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots$$

$$= \Theta(\text{largest term})$$

Let’s go back to our ThreeSum Algorithm

- We know that

$$T(n) = \frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0 + t_4x$$

- What is the order of growth of T(n)?
    - T(n) = O(n^3)

Let’s go back to our ThreeSum Algorithm

- Assuming that definition…
  - Is ThreeSum $O(n^4)$?
  - What about $O(n^5)$?
  - What about $O(3^n)$??
- If all of these are true, why was $O(n^3)$ what we jumped to to start?

Another mathematical notation

Tilde approximation (~)

- Same as Theta but keeps constant factors
- Two functions are Tilde of each other if they have the same order of growth and the same constant of the largest term

$$5n = \Theta(5{,}000{,}000{,}000\ n)$$

$$\not\sim 5{,}000{,}000{,}000\ n$$

A faster algorithm for 3-sum

- What if we sorted the array first?
  - For each pair of numbers, binary search for the third one that will make a sum of zero
    - e.g., a[i] = 10, a[j] = -7, binary search for -3
    - Be careful not to use the same number twice
- What is the runtime?
  - Still have two for-loops, but we replace the third with a binary search
    - What if the input data isn't sorted?
  - What about the sorting time?

$$\underbrace{n^2}_{\text{all pairs}} \underbrace{\log n}_{\text{Binary Search}} + \cancel{\underbrace{n \log n}_{\text{Sorting}}}$$

The 3-sum problem: can we do better?

- There is an O(n^2) algorithm
  - Idea 1: use hashing to find the third number
  - Idea 2: for each number, find the missing pair of numbers in linear time
- There is also an O(n log n) algorithm under special cases
- **Unsolved problem**: Is there a general O($n^{2-\varepsilon}$) algorithm for some $\varepsilon > 0$?